ACEEE Int. J. on Information Technology, Vol. 3, No. 1, March 2013

# Automatic Synthesis and Formal Verification of Interfaces Between Incompatible Soft Intellectual Properties

Fateh Boutekkouk[1]
[1] Department of Mathematics and Computer Science, University of Oum El Bouaghi, 04000, Algeria
fateh_boutekkouk@yahoo.fr

***Abstract*—In this work, we are concerned with automatic synthesis and formal verification of interfaces between incompatible soft intellectual properties (IPs) for System On Chip (SOC) design. IPs Structural and dynamic aspects are modeled via UML2.x diagrams such as structural, timing and Statecharts diagrams. From these diagrams, interfaces are generated automatically between incompatible IPs following an interface synthesis algorithm. Interfaces behaviors verification is performed by the model checker that is integrated in Maude language. A Maude specification including interface specification and properties for verification are generated automatically from UML diagrams.**

***Index Terms*—SOC, IP, Integration, UML, Maude, Formal verification**

## I. Introduction

Nowadays, Systems On Chip (SOC) [8] design is becoming more complex and may lead to the non satisfactory of customers requirements and the time to market constraints. To cope with this problem, it seems that Core Based Design (CBD) brings a significant improvement of design in general and to decrease the time to market window in particular [1, 9]. The main idea behind the CBD is to reutilize existing hardware and/or software components with some customization and adaptation.In the SOC field, designers have considered the reuse of complex hardware and software components (Intellectual Property or simply IP components), already used and tested in previous designs [6,7, 18, 23]. Reuse is essential to master the complexity of SOC design; however it does not come for free. Since most IPs are provided by different vendors, they have different interface schemes, data bit widths and operating frequencies, combining these components is an error-prone task. Designers have to find and evaluate IPs that fit particular needs and the selected IPs must be integrated together to implement the desired SOC functionality. This integration may require some adaptation and customization. The basic goal of an interface synthesis is to generate interfaces between incompatible components. For this reason, researchers in both academia and industry [2, 5, 10, 11, 14, 16, 17, 20, 21, 22] have developed many algorithms and CAD tools to explore, to optimize, and to generate interfaces between incompatible IPs. Unfortunately, most of these efforts target models and languages at lower levels of abstractions. Another problem is the fact that the generated interface may not work correctly. In this context, we have developed an UML 2.x [3, 4, 19] tool that permits to both software and hardware SOC designers to model, configure, and link the incompatible IPs graphically. From UML diagrams, a set of FSMDs (Finite State Machine with Data path) modeling the interface are generated automatically. Our tool permits also formal specification generation of the interface in the Maude language [12]. The rest of this paper is organized as follows: section two is dedicated to related works concerning the synthesis of interface for incompatible protocols. Section three gives an overview of IPs and their classes Section four puts the light on Maude language. The algorithm of interface synthesis we have adopted is detailed in section five. Section six discusses the translation from UML to Maude. Our developed tool is presented in section seven before conclusion.

## II. Related Work

The literature on interface synthesis is rich, here, we try to mention some pertinent works targeting interface generation. In [11], signal transition graph was introduced for protocol specification and the hardware interface is synthesized with asynchronous logic. In [14], the protocol specification is decomposed into basic operations, while the protocol is represented as an ordered set of relations whose execution is guarded by a condition or by a time delay. In [17], the two protocols are described using regular expressions and are translated into corresponding deterministic finite automata then interface protocol can be synthesized as an FSM by production computation algorithm. In [20, 21], a queue-based interface scheme was proposed. An algorithm which generates FSMD model for queue from timing specification of the given memory was developed. In order to generate the interface automatically, a formal model, called Protocol Sequence Graph (PSG) that captures the minimal necessary set of features representing the interface and its associated communication protocol. From given protocol specifications and clock period of the selected queue, the interface synthesis algorithm generates the FSMD for interface including the queue FSMD.

The main limitation of these approaches is that IPs communication protocols are expressed in low level models and/or programming languages such as waveforms, VHDL or C language. Another tendency to address the problem of IPs integration is the use of standards for promoting reuse in




DOI: 01.IJIT.3.1.1145

the design process. Several standards have been proposed.Among these, the Open Core Protocol (OCP) by OCP-IP [15] has gained wide industrial acceptance. However, for existing non OCP compliant IP cores, it is very expensive to customize them to comply with the OCP standard.

Our work tries to take advantages of the UML 2.x standard for IPs modeling and interface generation with minimal user inputs exploiting the algorithm proposed in [21]. In its basic form, this algorithm was used to generate the glue logic between two incompatible IPs. Since the system may contain many incompatible IPs, we have to apply the same algorithm for each pair of communicating incompatible IPs. Our tool differs from others in:

1. The use of high level models for incompatible IPs integration and in particular the use of UML 2.x new diagrams like timing and structure diagrams.

2. IPs communication protocols are abstracted from any Hardware Description Language (HDL) and specified using UML Statecharts where actions are associated to states and expressed in the C language.

3. Our tool supports both communication protocols customization and automatic interface generation. The generated glue logic is an FSMD modeled via UML Statecharts including concurrent and hierarchic states.

4. Our tool permits formal specification generation for interfaces between IPs in the Maude language.

Our choice of Maude language is due to its expressivity, simplicity, simulation, and formal verification at different levels of abstraction [12].

## III. Intellectual Property (IP)

An intellectual property or a virtual core (IP) [23] is a reusable software or hardware pre-designed block and maybe delivered by third party companies. Hardware IP components may come in several forms: hard, firm or soft. An IP is *hard*, when all its gates and interconnects are placed and routed. It has the advantage of more predictable estimations of performance, power, and area considering the target technology. But, it is less flexible and therefore less reusable. An IP can be *soft*, with only an RTL (Register Transfer Level) representation. It is available in source code and therefore adaptable to different platforms at the price of less predictable estimations on performance and area. An IP can be *firm*, with an RTL description together with some physical floor planning or placement.

## IV. Rewriting Logic and Maude Language

The rewriting logic was introduced by Meseguer [13]. This logic having a complete semantics unifies all the formal models that express concurrence. In rewriting logic, the logic formulas are called rewriting rules. They have the following form: R:[t] -> [t'] if C. Rule R indicates that term t is transformed into t' if a certain condition C if verified. Term represents a partial state of a global state S of the described system. The modification of the global state S of the system to another state S' is realized by the parallel rewriting of one or more terms that express the partial states. The distributed state of a concurrent system is represented as a term whose sub-terms represent the different components of the concurrent state.

Maude [12] is a specification and programming language based on the rewriting logic. Two specifications level are defined in Maude. The first level concerns the system specification, while the second one carries on the properties specification. The system specification level is provided by the rewrite theory. It is mainly specified by the system modules. For a good modular description, three types of modules are defined in Maude. Functional modules allow defining data types and their functions through equations theory.

The code bellow represents the functional module *Nat* specifying natural numbers. Here, we declare three sorts (types) that are *Zero*, *NzNat*, and *Nat*, and two functions that are *0* and *s*. Sorts *Zero* and *NzNat* are sub-sorts of the *Nat* sort. *0* is a special type of operation called a constructor (constant). To designate a function as a constructor, we add the keyword '*[ctor]*'. *Nat* module is imported in the module *FACT* to calculate the factorial of natural numbers. Modules importation is performed via *protecting*, *extending*, or *including*. Functions are implemented as equations through the keyword '*eq*'.

```
fmod NAT is
sorts Zero NzNat Nat .
subsort Zero NzNat < Nat .
***constructors
op 0 : -> Zero [ctor] .
op s_ : Nat -> NzNat .
….
endfm

fmod FACT is
Including NAT .
op _! : Nat -> NzNat .
var N : Nat .
eq 0 ! = 1 .
eq (s N) ! = (s N) * N !.
endfm
```

System modules define the dynamic behavior of a system. This type of modules extends functional modules by introducing rewriting rules. A maximal degree of concurrence is offered by this type of module. Finally, there are the object-oriented modules that can be reduced to system modules. In relation to system modules, object-oriented modules offer a more appropriate syntax to describe the basic entities of the object paradigm as, among others: objects, messages and configuration. Only one rewriting rule allows expressing the consumption of certain floating messages, the sending of new messages, the destruction of objects, the creation of new objects, state change of certain objects, etc.

The code above illustrates the use of a system module *BANK-ACCOUNT* to define an object counts banking *A* and the two operations capable to affect its content *credit* and *debit* while executing the rewriting rules defined in this





DOI: 01.IJIT.3.1. 1145

module. Note that after the execution of the unconditional rule *[credit]*, the message *credit(A, M)* is consumed and the content of the account is increased. In the same way, the execution of the conditional rule *[debit]* requires that the condition (N>=M) be verified. The execution of such rule generates the consumption of the message *debit(A,M)* and the reduction of the content of the account. The *BANK-ACCOUNT* module imports two predefined Maude modules that are *INT* for integers processing and *CONFIGURATION* denoting the subsets or soups of objects and messages.

```
mod BANK-ACCOUNT is
protecting INT .
including CONFIGURATION .
op Account : -> Cid.
op bal :_ : Int -> Attribute .
ops credit debit : Oid Nat -> Msg .
var A : Oid . vars M N :Int
rl [credit]: < A : Account | bal : N > credit(A, M) => < A :
Account | bal:N + M > .
crl [debit] : < A : Account | bal : N > debit(A, M) => < A :
Account | bal : N - M > If N >= M .
endm
```

The property specification level defines the system properties to be verified. The system is described using a system module. By evaluating the set of states reachable from an initial state, the model-checking allows to verify a given property in a state or a set of states. The Model-checking supported by Maude's platform essentially uses the LTL (Linear Temporal Logic) logic for its simplicity and the defined decision procedures it offers.

LTL operators are represented in Maude using a syntactic form similar to their original form. For example, the operation *[]* is defined in Maude by the operator (always). This latter is applied to a formula to give a new formula. Furthermore, we need an operator indicating if a given formula is true or false in a certain state. We find such an operator (|=) in a predefined module called SATISFACTION. The state *State* is generic. After specifying the behavior of its system in Maude system module, the user can specify several predicates expressing some properties related to the system. These predicates are described in a new module that imports in its turn, two modules: the first one that describes the system's dynamic aspect, where the second is the module SATISFACTION. Let, for example, M-PREDS the name of the module describing the predicates on the system's states. *M* is the name of the module describing the system's behavior. The user must specify that the chosen state (chosen configuration in this example) for its own system is sub-type of *State* type.

The code bellow describes a module in Maude implementing the operator of satisfaction of a formula in a state.

```
fmod SATISFACTION is
protecting LTL .
 sort State .
 op _|=_ : State Formula ~> Bool .
endfm
```

The code bellow describes a module in Maude containing predicates defined by the user about a system described by a module M.

```
mod M-PREDS is
 protecting M .
 including SATISFACTION .
 subsort Configuration < State .
 …
endm
```

At the end, we find the MODEL-CHECKER module that offers the function *model-Check*. The user can call this function while specifying a given initial state and a formula. Maude model-Checker verifies if this formula is valid (according to the nature of the formula and the procedure of the model-checker adopted by Maude system) in this state or the set of all reachable states form the initial state. If the formula is not valid, a counter example (counterexample) is displayed. The counter example concerns the state in which the formula is not valid.

The code bellow describes a module in Maude containing the services offered to the user by Maude Model-checker.

```
fmod MODEL-CHECKER is
including SATISFACTION .
…
op counterexample : TransitionList TransitionList ->
ModelCheckResult [ctor] .
op modelCheck : State Formula ~> ModelCheckResult .
…
Endfm
```

## V. Interface Generation Algorithm

In this section, we try to put the light on the synthesis algorithm for interface between incompatible protocols as proposed in [21]. In [20], the interface architecture is basically composed of synchronous system interfaces as shown in Figure 1. The system components (PE1 and PE2) may operate at different frequencies and at different data rates. The interface architecture includes a buffer (FIFO queue) to smoothen the burst data transfer requests and two FSMDs (Finite State Machine with Data path) to queue and un-queue data. In the interface architecture, system components (PE1 and PE2) in Figure 1 are directly connected to its corresponding state machines and will transfer data to other component through the state machines. The state machines are responsible for receiving (sending) data from (to) the corresponding system components and writing (reading) the data to (from) the queues. We have to consider two interface protocols, the protocol between state machines and queues and the protocol between system components and state machines. The interface protocol between state machines and queues will be fixed because the queue interface is predefined. But the interface protocol between system components and state machines will be varied depending on the protocol of system components.The queue is implemented with a memory to store large amount of data. The clock period of the queue is frequently less than the memory read access time.Generally, a queue contains memory




DOI: 01.IJIT.3.1.1145

to store data internally. The operation of the queue is determined by memory organization and timing [20]. In order to generate a queue model from the memory timing constraints, we have to schedule the timing constraints based on given clock period of the queue. Given timing constraints of the memory and the clock period of the queue, queue generation reduces to the task of generating a state machine that implements the given queue functionality and satisfies the timing constraints. This requires scheduling of memory timing constraints into clock cycles such that no constraint is violated. Therefore, the FSMD implementation selects instances of the given timing ranges based on the granularity given by the queue clock. Finally the queue description will be generated for integration in interface synthesis. Figure 2 shows the block diagram of a queue with a single I/O port.

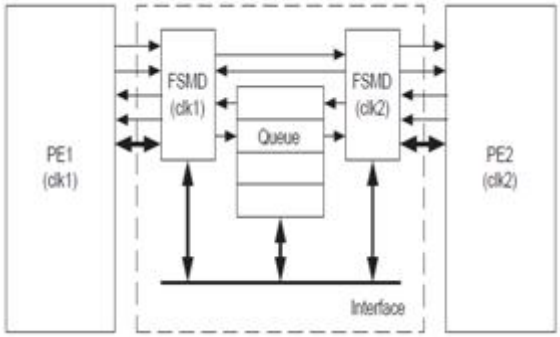


Fig. 1. The interface architecture of two incompatible Components [9]

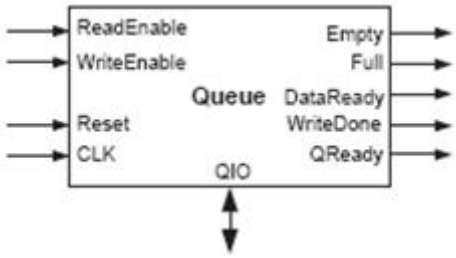


Figure 2. Queue with a single I/O port [9]

*A. Interface Generation Algorithm*

*Problem definition*

*Given:*

1. Protocol descriptions of two communicating parties (producer and consumer).
2. Bit width and size for the selected memory.
3. Clock period *TQclk* of the queue.

*Determine:*

1. FSMDs for state machines.
2. FSMD for the queue.

*Conditions:* Timing constraints are met.

Algorithm of the figure 3 shows the interface synthesis algorithm from given protocol specifications and clock period of the selected queue. We have applied the same algorithm as proposed in [21] but with two major modifications: firstly, the scheduling of actions over states is performed by the designer thus the *Schedule()* function is removed from the algorithm. Secondly, the *Make_Dual()* function transforms the Statechart (instead of the Protocol sequence graph PSG) of the original protocol specification to the corresponding dual Statechart, which can be done by replacing the operators in actions with their duals.

The method *Generate_Queue()* will generate the queue. The generated producer interface FSMD, consumer interface FSMD and queue interface FSMD should be collapsed into a single FSMD to obtain interface FSMD. The method *Add_FSMD()* will collapse the producer and queue interface FSMDs(*FMSDSi*, *FSMDQi*) into the transducer interface FSMD for the producer (*FSMDTS*). In the same way, the consumer and queue interface FSMD(*FSMDRi*, *FSMDQi*) will collapse into the transducer interface FSMD for the consumer. Finally we have two FSMDs for transducer: the producer interface FSMD and the consumer interface FSMD in the transducer. For more details on this algorithm, one can refer to [21].

```
Algorithm GenerateInterface ( FSMD_S , FSMD_R , T_Qclk )
FSMD_Q = Generate_Queue(T_Qclk);   // generate Queue FSMD
FSMD_Si = Make_Dual(FSMD_S); // generate the dual of producer
FSMD_Ri = Make_Dual(FSMD_R); // generate the dual of consumer
FSMD_Qi = Make_Dual(FSMD_Q ); // generate the dual of queue
For i = 1 to ( bw_Q / bw_S ) do
  Add_FSMD(FSMD_TS, FSMD_Si); // add the producer FSMD to interface FSMD
End for
  Add_FSMD(FSMD_TS, FMSD_Qi); // add the queue FSMD to producer interface FSMD
  Add_FSMD(FSMD_TR, FMSD_Qi); // add the queue FSMD to consumer interface FSMD
For i = 1 to ( bw_R / bw_Q ) do
  Add_FSMD(FSMD_TR, FSMD_Ri); // add the consumer FSMD to interface FSMD
End for
```

Fig. 3. The generate interface algorithm

The generate queue problem is presented as follows:

Problem definition

Given:

1. Timing parameters *Tacc*, *Toh*, *Tas*, *Twpw*, and *Tah* for selected memory.
2. Size (bit width and depth) of the selected memory
3. Clock period *Tclk* of the queue.

*Determine:* Finite state machine with data model for the queue.

*Condition:* Memory timing constraints are satisfied.

Algorithm of the figure 4 describes the queue generation algorithm from given memory timing constraints and clock period of the queue. The function Add_State(FSMD, S) adds state S into state machine (FSMD). First, function Generate_Reset_State generates reset state (S0), in which every output signal and internal variables for the memory and counters are initialized. Whenever reset is asserted, the state of queue is in this state. Function Generate_Initial_State generates initial state (S1), in which all output signals are de-asserted until ReadEnable or WriteEnable gets asserted by external producer and consumer. For read cycle operation, memory access state (S2) is generated according to memory





DOI: 01.IJIT.3.1.1145

```
Algorithm Generate _Queue ( Tacc , Toh , Tas , Twpw , Tah )
{ S0 = Generate_Reset_State ();
S1 = Generate_Initial_State ();
Add_State(Read_States , S0);
Add_State(Read_States , S1);
For i =1 to [ Tacc / TCLK ] do
{ S2=Generate_Mem_Access_State (); // generate read states
Add_State(Read_States , S2); }
S3 = Generate_Data_Ready_State ();
Add_State(Read_States , S3);
For i =1 to [ Toh / TCLK ] do
{S4=Generate_Mem_Output_Hold_State ();
Add_State(Read_States , S4); }
For i=1 to [ Tas / TCLK ] do
{S2=Generate_Mem_Address_Setup_State ();
Add_State(write_States , S2); }
For i=1 to [ Twpw / TCLK ] do
{ S3=Generate_Mem_Write_State (); // generate write states
Add_State(write_States , S3); }
For i =1 to [ Tah / TCLK ] do
{S4=Generate_Mem_Address_Setup_State ();
Add_State(write_States , S4); }
S5=Generate_Write_Stat_Done ();
Add_State(write_States , S5);
```

Fig. 4. The generate interface algorithm

access time *Tacc* in function Generate_Mem_Access_State, and data ready state (S3) by functionGenerate_Data_Ready _State. Finally, function Generate_Mem_Output_Hold_State generates memory output hold state (S4) based on output hold time *Toh*. For write cycle operation, memory address setup state (S2) is generated according to memory address setup time *Tas* in function Generate_Mem_Address_Setup _State.

Function Generate_Mem_Write_State generates the memory write state (S3) according to write pulse width time *Twpw*.Function Generate_Mem_Address_Hold_State generates memory address hold state (S4), based on output hold time *Tah*. Finally, the memory write done state (S5) is generated by function Generate_Mem_Write_Done_State [20].

## VI. Passage from UML to Maude

### A. Translation of Static Aspects

As an example of application, we have chosen three IPs that are: *ColdFire* processor, *ARM9TDMI* processor, and *TMS320C50 DSP* processor [21]. The objective is to generate and verify the interfaces between these three cores with incompatible communication protocols. For more detail on this example, one can refer to [21]. In this section we will explain the translation from UML to Maude specifications.
UML objects are specified as Maude objects (class instances), so for each IP, we declare a class with a set of attributes. Input/output interfaces (signals) are specified as Maude attributes. Signals generation or assignments are specified as Maude messages. In the bellow example, we declare an object called *Pr*.
*<Pr : Arm9tdmi | DA : addr, state : s1, DD : data, DnRW : M, DDEN : N, nWAIT : X > Pr* is an instance of the class *Arm9tdmi* representing the *Arm9tdmi* IP. In Maude, we declare the *Arm9tdmi class* as follows: *op Arm9tdmi : -> Cid [ctor] .*
*Pr* has five signals that are *DA*, *DD*, *DnRW*, *DDEN*, and *nWait*. In Maude, we declare a signal as follows:
*op DA :_ : Int -> Attribute [ctor gather (&)] .*

We add a new attribute called *state* to specify the current state of the IP. For this we declare a new sort (type) called *statevalues*. Possible values for this type are all possible states of the IP. In Maude, we write:
*sort statevalues .*
*ops s1 s2 r1 r2 e1 e2 e3 : ->statevalues .*
Similarly, we specify a FSMD for instance as:
*<Fsmd1 :Fsmd | state : q0, Full : N1, QReady : Y, WriteEnable : V >*

### B. Translation of Dynamic Aspects

FSMD transitions are specified as Maude rewriting rules. On a transition, we can find incoming/outcoming events that correspond to signals reading/writing. Such events are specified in Maude as messages. Here is an example of a rewriting rule:
*rl [rl1] :*
*signaldone(Pr, A)*
*signalmask(Pr, G)*
*signalinport(Pr, data)*
*<Pr : Arm9tdmi | state : e1, inport : data, mask : G, done : A >*
*=> signalDD(Pr, 5) signalDA(Pr, 1000) signalDnRW(Pr, 1) signalnWAIT(Pr, 1) signalDDEN(Pr, 1)*
*<Pr : Arm9tdmi | DA : 1000, state : s1, DD : 5, DnRW : 1, DDEN : 1 > .*

This unconditional rule enables the IP *Pr* to transit from the state *e1* to the state *s1*and to modify the values of signals *DA*, *DD*, *DnRW*, and *DDEN*.
*signaldone(Pr, A)* is a message that specifies an input event ( the value of *signaldone* is true).
*signalDD(Pr, 5)* is a message that specifies an output event (we assign the value 5 to signal *signalDD* ).
Table 1 shows the correspondence between UML and Maude constructors.

Table I. Correspondence Between Uml And Maude

| UML | Maude |
|---|---|
| Composite object | System module |
| Simple object | Object (Class instance) |
| State diagram transition | Rewriting rule |
| Required/Provided interfaces | Attributes |
| Hierarchic state | Flat specification |
| Concurrent state | A set of rewriting rules |
| Input/Output events on transitions | Messages |

The code bellow defines the Maude initial configuration of our system including ARM9TDMI, Coldfire processors,




DOI: 01.IJIT.3.1.1145

FSMD1 and FSMD2.

*op Arm9 : ->Oid .*
*op Cold : ->Oid .*
*op fsm1 : ->Oid .*
*op fsm2 : ->Oid .*
*ops Initial Initial1 Initial2 Initial3 Initial4 Final1 Final2 Final3 Final4 : -> Configuration .*
*eq Initial = signaldone(Arm9, 0) signalmask(Arm9, 1) signalinport(Arm9, 5) < Arm9 : Arm9tdmi | state : e1, inport : 5, mask : 1, done : 0 >*
*signalDA(fsm1, 1000) signalDD(fsm1, 5) signalDnRW(fsm1, 1) signalnWAIT(fsm1, 1) signalDDEN(fsm1, 1) < fsm1 : Fsmd | DA : 1000, stateFSMD1 : s1, DnRW : 1, DDEN : 1, nWAIT : 1 >*
*signalRE(fsm2, 0) signalr2(fsm2, 0) signalacr1(fsm2, 0) signalE(fsm2, 0) signalQR(fsm2, 1) signalr1(fsm2, 1) < fsm2 : Fsmd | ReadEnable : 0, stateFSMD2 : q0, ready2 : 0, ackready1 : 0, Empty : 0, QReady : 1, ready1 : 1 >*
*< Arm9 : Arm9tdmi | state : e2, done : 1, outport : 1 >< fsm1 : Fsmd | DA : 0, stateFSMD1 : r2, DnRW : 1, DDEN : 0 >< fsm2 : Fsmd | stateFSMD2 : q5, ready2 : 0 >< Cold : Coldfire | state : s2, MAPB : 1, MDPB : 1, MRWB : 1 > .*
*eq Initial1 = signaldone(Arm9, 0) signalmask(Arm9, 1) signalinport(Arm9, 5) < Arm9 : Arm9tdmi | state : e1, inport : 5, mask : 1, done : 0 > .*
*eq Initial2 = signalDA(fsm1, 1000) signalDD(fsm1, 5) signalDnRW(fsm1, 1) signalnWAIT(fsm1, 1) signalDDEN(fsm1, 1) < fsm1 : Fsmd | DA : 1000, stateFSMD1 : s1, DnRW : 1, DDEN : 1, nWAIT : 1 > .*
*eq Initial3 = signalRE(fsm2, 0) signalr2(fsm2, 0) signalacr1(fsm2, 0) signalE(fsm2, 0) signalQR(fsm2, 1) signalr1(fsm2, 1) < fsm2 : Fsmd | ReadEnable : 0, stateFSMD2 : q0, ready2 : 0, ackready1 : 0, Empty : 0, QReady : 1, ready1 : 1 > .*
*eq Initial4 = signalMDPB(Cold, 5) signalMAPB(Cold, 1) signalMRWB(Cold, 1) signalMADDR(Cold, 1000) < Cold : Coldfire | MADDR : 1000, state : r1, MAPB : 1, MDPB : 5, MRWB : 1 > .*
*eq Final1 = < Arm9 : Arm9tdmi | state : e2, done : 1, outport : 1 > .*
*eq Final2 = < fsm1 : Fsmd | DA : 0, stateFSMD1 : r2, DnRW : 1, DDEN : 0 > .*
*eq Final3 = < fsm2 : Fsmd | stateFSMD2 : q5, ready2 : 0 > .eq Final4 = < Cold : Coldfire | state : s2, MAPB : 1, MDPB : 1, MRWB : 1 > .*

### *C. Specification of the Interface Properties*

Two properties are defined: the *begin* property that specifies the fact that the interface FSMDs are in their beginning states. The *end* property specifies the fact that the interface FSMDs are in their end states. In Maude, we declare these two properties as follows:

*op begin : Configuration Configuration Configuration Configuration -> Prop .*
*op end : Configuration Configuration Configuration Configuration -> Prop .*

Bellow, the definition of the *end* property in Maude:

*ceq< Pro : Arm9tdmi | state : S, done : A, outport : G1 >Cf <Fsmd1 :Fsmd | DA : addr, stateFSMD1 : S1, DnRW : M2, DDEN : N2 > Cf <Fsmd2 :Fsmd | stateFSMD2 : S2, ready2 : Z4 > Cf <Pro1 :Coldfire | state : S3, MAPB : adr, MDPB : dat, MRWB : W > Cf |= end(< Pro : Arm9tdmi | state : S, done : A, outport : G1 >Cf, < Fsmd1 : Fsmd | DA : addr, stateFSMD1 : S1, DnRW : M2, DDEN : N2 >Cf, < Fsmd2 : Fsmd | stateFSMD2 : S2, ready2 : Z4 >Cf, < Pro1 : Coldfire | state : S3, MAPB : adr, MDPB : dat, MRWB : W >Cf) = true if S == e2 /\ S1 == r2 /\ S2 == q5 /\ S3 == s2 .*

We want to verify reachability: do interface FSMDs always reach *end* states from *begin* states ?

In Maude, we specify such query as follows:

*[] begin(Initial1,Initial2,Initai3,Initial1) -> end(Final1,Final2,Final3,Final4))*

*[]* is the always operator; -> is the implication operator.

In order to perform formal verification, we call the *ModelChecker* function as follows:

*"ModelCheck( [] begin(Initial1,Initial2,Initai3,Initial1) -> end(Final1,Final2,Final3,Final4))".*

## VII. Presentation of Our Tool

We have developed a tool that supports UML 2.x modeling of IPs using Structure, Timing, and Statecharts diagrams. Our tool generates automatically the interface FSMD including the Queue FSMD for each pair of communicating IPs. From Memory Timing diagram, some temporal parameters are extracted to be used for FSMD queue generation. In our case, each IP is modeled via UML 2.x components with required (input) and provides (output) signals.

Furthermore, each IP is parameterized by some parameters such as the HDL (e.g. VHDL, Verilog), the clock period, the power consumption, and the abstraction level of each IP. Some of these parameters (Power consumption) are not used in the algorithm of interface synthesis, rather than, we will use them to perform power estimation in our future work. In our case and regardless of the IP HDL, we assume that all IPs communication protocols are modeled via UML Statecharts. The communication protocols actions are expressed in the C language. Figure 5 shows IPs modeling and connections between them. Figure 6 shows timing parameters modeling of memory read/write cycles.

Figure 7 shows IP internal behavior modeling. Figure 8 shows the interface FSMD. The latter is generated automatically.

Figure 9 shows the generated Maude code for the interface. Maude Properties code is introduced by the designer and it is automatically added to the interface code to finally generate one Maude file including both interface code and properties.

Figure 10 and 11 show respectively the results of rewriting rules execution (behavior simulation) for individual behavior of ARM9TDMI and COLDFIRE processors .

Figure 12 shows individual FSMD1 and FSMD2 behavior simulation results. Figure 13 shows the collective interface behavior simulation result.




DOI: 01.IJIT.3.1. 1145

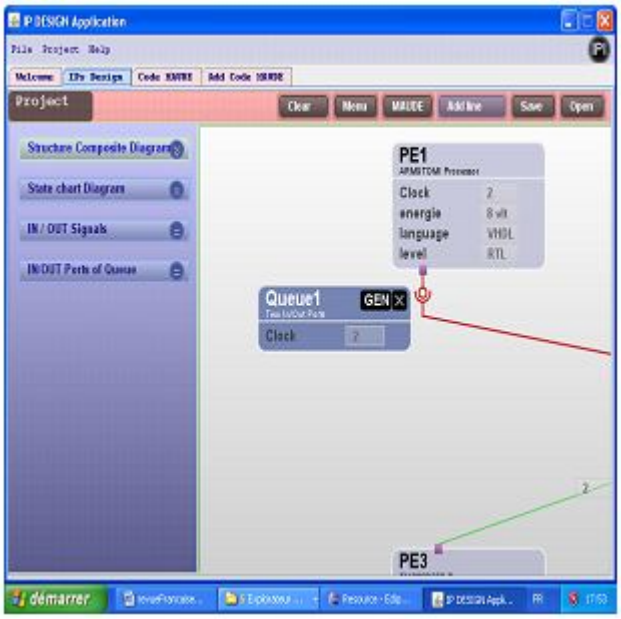


Fig. 5. UML modeling of IPs and their connections


Fig. 6. Timing parameters modeling


Fig. 7. IP behavior modeling


Fig. 8. UML modeling of the automatic generated Interface

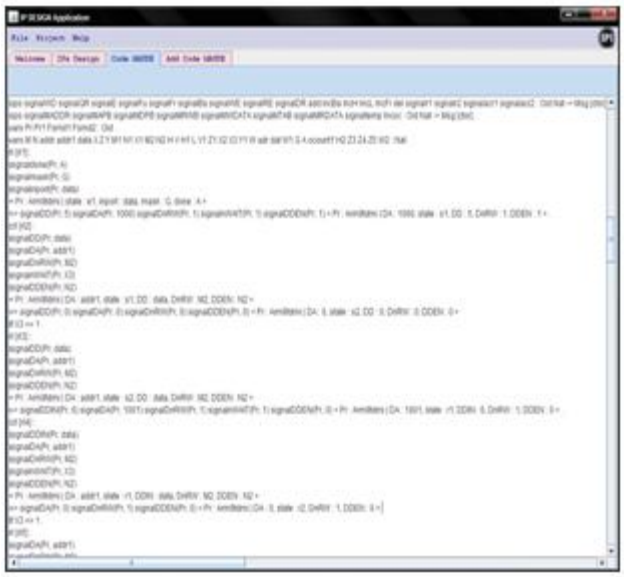

Fig. 9. Maude code automatic generation for the interface

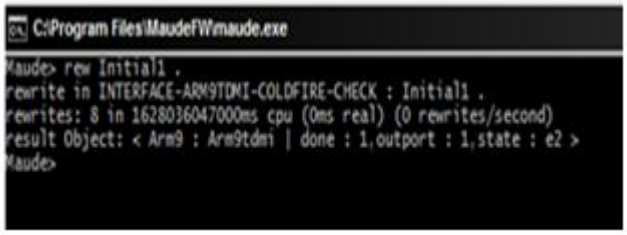


Fig. 10. ARM9TDMI behavior simulation

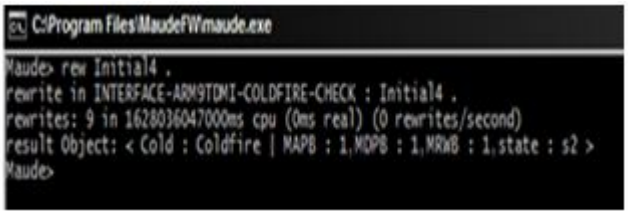


Fig. 11. COLDFIRE behavior simulation




DOI: 01.IJIT.3.1.1145

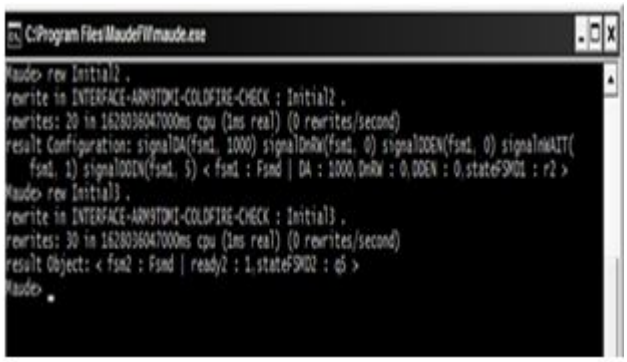


Fig. 12. FSMD1 and FSMD2 behaviors simulation

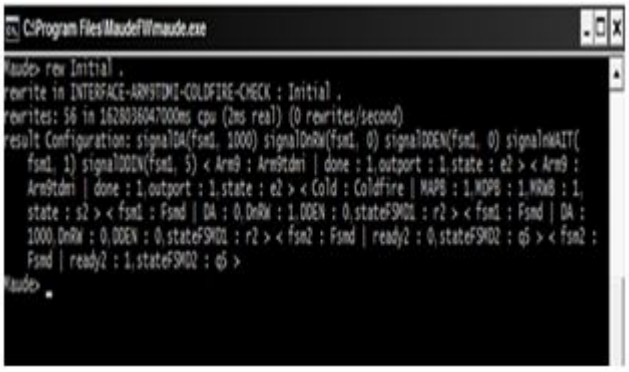


Fig. 13. FSMD1 and FSMD2 behaviors simulation

Figure 14 shows the result of model checker for the reachability property between the two IPs . *ARM9TDMI* and *ColdFire* processors. According to the model checker result, we can state that the reachability property is verified.


Fig. 14. Model checker execution

## Conclusion and Perspectives

In this work, we presented our tool for IPs modeling using UML 2.x diagrams and interface formal verification between incompatible IPs using Maude language. Such a tool will help SOC designers to integrate incompatible IPs by the generation, simulation and formal verification of interfaces. We have exploited UML 2.x diagrams such as structure diagram for modeling and connection between IPs, timing diagram to model memory read and write operations timing constraints, and Statecharts with hierarchic and concurrent states to model IPs communication protocols and Queue behavior. Using graphical notations offered by UML brings more conviviality especially for software designers. Formal verification is performed by calling the model checker which is integrated in the Maude formal language. As a perspective, we plan to discover more properties for verification and to perform performance and power consumption estimation on the generated interface.

## References

[1] J. Biggs and A. Gibbon, « Reference Methodology for Enabling Core Based Design », European Synopsys User Group, SNUG, March 2002.

[2] G. Borriello and R. Katz, « Synthesis and optimization of interface transducer logic », Proceedings of the International Conference on Computer-Aided Design, November 1987, p. 274–277.

[3] G. Booch, J. Rumbaugh, and I. Jacobson, Unified Modeling Language User Guide. Addison-Wesley, 1999.

[4] F. Boutekkouk, M. Benmohammed, S. Bilavarn, and M. Auguin, « UML 2.0 profiles for Embedded Systems and Systems On a Chip (SOCs) », In JOT: Journal of Object Technology , January 2009.

[5] K.S. Chung, R.K. Gupta, and C.L. Liu, « An algorithm for synthesis of system level interface circuits », In ICCAD 96. Digest of Technical Papers. 1996 IEEE/ACM International Conference ,10-14 November 1996, p. 442-447.

[6] P.L. Flake, S. J. Davidmann, D.J. Kelf, and C. Burisch, « The IP Reuse Requirements for System Level Design Languages », International Property 2000 Conference, Santa Clara, CA, April 2000.

[7] D. Gajski, F. Vahid, S. Narayan, and J. Gong, Specification and Design of Embedded Systems, published by Prentice Hall. Englewood, New jersey 07632, 1994.

[8] A.A. Jerraya and W. Wolf, Multiprocessor systems on chip, Morgan Kaufmann publishers, 2005.

[9] M. Keating and P. Bricaud, Reuse Methodology manual for systems-on-chip designs. 3rd edition, Kluwer Academic publishers, 2002.

[10] T. Kun, H. Wang, and J.N. Bian, « A generic interface modelling approach for SOC design », in ICSICT'04, , Beijing, China, 2004, p.1400-1403.

[11] B. Lin and S. Vercauteren, « Synthesis of concurrent system interface modules with automatic protocol conversion generation », Proceedings of the International Conference on Computer-Aided Design, November 1994, p. 101–108.

[12] T. McCombs, Maude 2.0 Primer, Version 1.0, International report. SRI. International, 2003.

[13] J. Meseguer, « Rewriting as a unified model of concurrency », proceedings of the Concurr'90 Conference, Springer LNCS, Amsterdam , vol. 458, 1990, p. 384-400.

[14] S. Narayan and D. Gajski, « Interfacing incompatible protocols using interface process generation », Proceedings of the Design Automation Conference, November 1994, p. 468–473.

[15] OCP-IP, http://www.ocp-ip.org.

[16] B. Park, H. Choi, I.C. Park, and C.M. Kyung, « Synthesis and optimization of interface hardware between ip's operating at





DOI: 01.IJIT.3.1.1145

different clock frequencies », in Proceedings of the International Conference on Computer Design, pages 519–524, June 2000.

[17] R. Passerone, J. A. Rowson, and A. Sangiovanni-Vincentelli, « Automatic synthesis of interfaces between incompatible protocols », Proceedings of the Design Automation Conference, June 1998, p. 8–13.

[18] T. Roudier, I. Moussa, and P. Crescenzo, « IP Modelling and Reuse for System Level Design », published for DATE 2003.

[19] T. Schattkowsky, « UML2.0 Overview and Perspectives in SOC Design », Proceedings of the Design, Automation and Test in Europe (DATE'05).

[20] D. Shin and D. Gajski, Queue Generation Algorithm for Interface Synthesis, Technical Report ICS-TR-02-03, University of California, Irvine, February 2002.

[21] D. Shin and D. Gajski, Interface synthesis from protocol specification, Technical Report CECS-02-13, University of California, Irvine, April 12, 2002.

[22] J. Smith and S. DeMicheli, « Automated composition of hardware components », Proceedings of the Design Automation Conference, June 1998, p. 14–19.

[23] F. R. Wagner, W. O. Cesario, L. Carro, and A.A. Jerraya, « Strategies for integration of hardware and software IP components in embedded systems on chip », VLSI journal, Elsevier, 2004.


DOI: 01.IJIT.3.1.1145